Bab 8

Query (III): Topik Lain-Lain

Bab ini secara sekilas membahas beberapa topik penting lainnya yang perlu Anda ketahui seputar query SQL, sebelum kita beranjak ke Bab 9 yang akan memperlihatkan berbagai contoh kasus.

8.1 View

View adalah sebuah tabel virtual yang dibentuk dari perintah SELECT:

```
-- Resep 8-1-1: Sintaks CREATE VIEW
CREATE VIEW nama_view AS
  SELECT ...;
```

Setelah dibuat, sebuah view dapat dipakai sebagaimana layaknya sebuah tabel, yaitu di-SELECT: `SELECT FROM view`. Sebagian view bisa juga di-INSERT atau di-UPDATE, namun tidak semua view bersifat updatabale. View-view tertentu misalnya yang mengandung klausa GROUP BY tidaklah bisa diupdate karena sebuah baris di view merupakan kumpulan beberapa baris dari tabel sumbernya.

Untuk menghapus VIEW, digunakan DROP VIEW.

Manfaat view adalah untuk mempersederhana query. Misalnya query yang kompleks bisa diselesaikan dulu setengah jalan dan ditaruh sebagai view. Lalu kita tinggal melakukan select yang lebih sederhana pada view yang bersangkutan. Manfaat kedua adalah untuk keamanan. Sebuah view dapat diberi permission yang berbeda dari tabel sumbernya. Jadi kita bisa mengambil sebagian baris saja atau sebagian kolom saja dari sebuah tabel yang bersifat privat (misalnya kolom *gaji* atau *password* tidak diikutsertakan) dan membuka view untuk publik.

Saat ini VIEW belum didukung MySQL. Namun karena pada dasarnya view adalah subselect yang diberi nama, kita bisa juga menggunakan subselect, mis: `SELECT FROM (SELECT FROM ...)`.

8.2 Trigger dan Stored Procedure

Stored procedure adalah sebuah fungsi atau prosedur dalam bahasa SQL prosedural yang dapat kita buat sendiri. Sementara trigger adalah sebuah kode SQL prosedural atau sebuah fungsi buatan user yang dapat dipanggil manakala sebuah baris ingin ditambahkan/diubah/dihapus dari sebuah tabel. Trigger dapat memanggil stored procedure.

Manfaat trigger amat banyak karena trigger sifatnya generik. Misalnya, trigger dapat dipakai untuk mengimplementasikan semua jenis constraint, misalnya: gagalkan INSERT jika INSERT tersebut mengandung nilai NULL (mengimplementasi constraint NOT NULL), atau gagalkan DELETE jika ternyata nilai kolom masih

dirujuk di tabel lain (mengimplementasi foreign key). Jika kita ingin constraint yang lebih kompleks daripada sekedar foreign key, UK, PK, atau jenis constraint lainnya, maka kita harus menggunakan bantuan trigger.

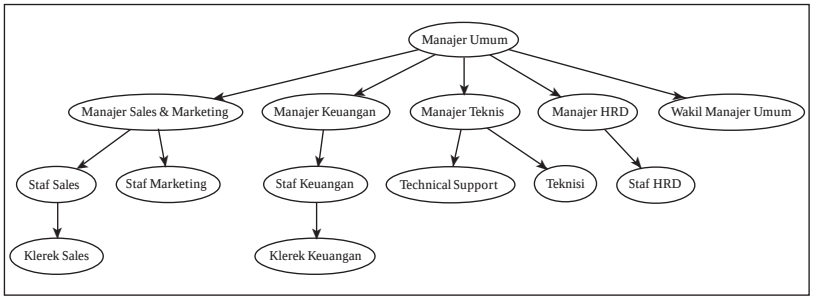
Di luar constraint, trigger juga bisa kita pakai untuk mengirim email jika ada baris tertentu yang ditambahkan ke sebuah tabel. Atau mengirim SMS, atau melakukan logging, dan lain sebagainya.

Umumnya stored procedure dipanggil dari trigger, namun dapat juga dipanggil manual menggunakan perintah SQL EXEC.

Sayangnya, saat ini trigger belum didukung di MySQL. Sementara stored procedure baru muncul di versi alfa MySQL (5.0) dan masih dalam pengembangan.

Dengan stored procedure kita bisa banyak menulis logika bisnis di dalam database.

8.3 Tree



Gambar 8-3-1.

Tree atau pohon atau struktur hirarkis seringkali dijumpai dalam data kita, misalnya dalam struktur organisasi, kategori produk, kategori link di direktori Web, atau dalam bills of material. Sebuah pohon terdiri dari node-node (titik-titik) yang dihubungkan dengan edge. Edge bersifat satu arah, yaitu dari parent node (induk) ke child node (anak). Setiap node dapat memiliki nol atau lebih child node. Di struktur teratas terdapat node yang tidak menjadi child node dari siapa pun, dan ini disebut root node.

Tidak seperti model hirarkikal (misalnya: XML) yang memang langsung memodelkan data dalam bentuk pohon, model relasional agak sedikit 'canggung' memodelkan struktur pohon ini. Namun bukan berarti tidak bisa, karena model relasional cukup fleksibel dan generik.

8.3 Tree

Ada beberapa cara umum yang dipakai orang untuk menyimpan tree: adjacency list, nested set, dan materialized path.

Adjacency List

Adjacency list adalah cara di mana kita menyimpan PK dari parent node di setiap node. Contoh:

```
CREATE TABLE tree (
  id INT PRIMARY KEY,
  parent_id INT NOT NULL
);
```

Root node dapat dinyatakan dengan baris yang parent_id-nya 0 (di mana semua non-root node nilai `parent_id`-nya tidak 0). Untuk menjaga integritas tree, kita dapat menambahkan trigger untuk mewajibkan agar nilai *parent_id* ada di dalam kolom *id* tree *atau* bukan diri sendiri *atau* atau 0. Trigger juga dapat menjaga agar hanya boleh ada satu baris yang *parent_id*-nya 0 (sehingga tabel *tree* hanya bisa menyimpan satu buah tree saja). Ini tidak bisa dilakukan dengan foreign key constraint biasa atau dengan kombinasi constraint CHECK lainnya dikarenakan sifat constraint yang berupa *atau*. Jadi *id* pada *parent_id* harus merupakan *id* yang valid pada tabel tree *atau* 0.

Karena MySQL tidak menyediakan trigger, maka kita harus merelakan tidak adanya penjaga integritas. Aplikasi Anda harus berhati-hati agar tidak memasukkan sembarang nilai *parent_id* ke dalam database, agar struktur pohon tidak rusak.

Berikut ini struktur organisasi pada Gambar 8-3-1 dinyatakan dengan model adjacency:

```
CREATE TABLE org_adjacency (
  id INT PRIMARY KEY,
  parent_id INT NOT NULL, INDEX(parent_id),
  nama VARCHAR(32)
);
mysql> SELECT * FROM org_adjacency;
+----+-----------+--------------------------+
| id | parent_id | nama                     |
+----+-----------+--------------------------+
|  1 |         0 | Manajer Umum             |
|  2 |         1 | Manajer Sales & Marketing |
|  3 |         1 | Manajer Keuangan         |
```

```
|  4 |         1 | Manajer Teknis            |
|  5 |         1 | Manajer HRD               |
|  6 |         1 | Wakil Manajer Umum        |
|  7 |         2 | Staf Sales                |
|  8 |         2 | Staf Marketing            |
|  9 |         7 | Klerek Sales              |
| 10 |         3 | Staf Keuangan             |
| 11 |        10 | Klerek Keuangan           |
| 12 |         4 | Technical Support         |
| 13 |         4 | Teknisi                   |
| 14 |         5 | Staf HRD                  |
+----+-----------+---------------------------+
14 rows in set (0.08 sec)
```

Dengan model ini, bagaimana cara mencari child node dari sebuah node *X*? Misalnya, siapa yang berada langsung di bawah Manajer Umum?

```
SELECT ... FROM tree WHERE parent_id=X;

-- contoh (di sini kita mengambil dulu id Manajer Umum via
subselect)
mysql> SELECT id, nama FROM org_adjacency WHERE parent_id=
        (SELECT id FROM org_adjacency WHERE nama='Manajer
Umum');
```

Namun untuk mencari semua descendant nodes dari node *X* (child node + child dari child node + dan seterusnya) dibutuhkan query terpisah untuk setiap levelnya:

```
-- mencari level pertama, child node (anak)
SELECT ... FROM tree WHERE parent_id=X
UNION
-- mencari level kedua, child dari child node (cucu)
SELECT ... FROM tree WHERE parent_id IN
  (SELECT id FROM tree WHERE parent_id=X)
UNION
-- mencari level ketiga (buyut)
SELECT ... FROM tree WHERE parent_id IN
  (SELECT id FROM tree WHERE parent_id IN
    (SELECT id FROM tree WHERE parent_id=X))
-- dan seterusnya
[UNION ...];
```

Contohnya, cari semua yang ada di bawah Manajer Sales & Marketing hingga 2 level:

```
-- contoh (di sini kita asumsikan kita telah memperoleh id Manajer
--   Sales & Marketing, yaitu 2)
mysql> SELECT id, nama FROM org_adjacency WHERE parent_id=2
       UNION
       SELECT id, nama FROM org_adjacency WHERE parent_id IN
         (SELECT id FROM org_adjacency WHERE parent_id=2);
+----+---------------+
| id | nama          |
+----+---------------+
|  7 | Staf Sales    |
|  8 | Staf Marketing |
|  9 | Klerek Sales  |
+----+---------------+
3 rows in set (0.06 sec)
```

Bisa dibayangkan repotnya jika ingin mencari hingga 10 level; atau hingga level yang tak terbatas dalamnya.

Untuk memasukkan sebuah child node untuk node *X*, caranya amat sederhana, cukup lakukan INSERT:

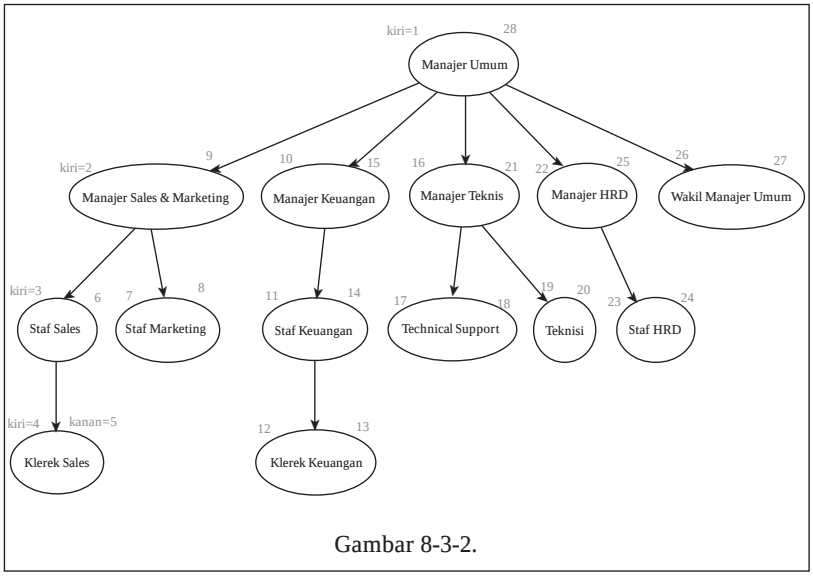
```
INSERT INTO tree (parent_id, id, ...) VALUES (X, ..., ....);
```

Untuk menghapus sebuah node beserta semua descendant nodesnya, lagi-lagi kita harus melakukan query yang berulang-ulang untuk setiap levelnya seperti sewaktu mencari subnodes.

```
DELETE FROM tree WHERE id IN
  (SELECT id FROM tree WHERE id=X – hapus diri sendiri
   UNION
   SELECT id FROM tree WHERE parent_id=X – hapus anak-anak
   UNION
   SELECT id FROM tree WHERE parent_id IN – hapus cucu
     (SELECT id FROM tree WHERE parent_id=X)
   [...]
  );
```

Nested Set

Metode ini menyimpan node beserta dua buah angka (yang dapat berupa bilangan bulat atau bilangan pecahan desimal) yang disebut *kiri* dan *kanan*. Angka pada kiri dan kanan disusun sedemikian rupa sehingga jika sebuah node nilai *kiri* dan *kanan*-nya lebih lebar dan mencakup *kiri* dan *kanan* milik node lain (*kiri* lebih kecil, kanan lebih besar), maka itu berarti node pertama disebut mencakupi (berada sebagai parent dari node kedua). Contohnya, struktur organisasi di Gambar 8-3-1 dapat disimpan dalam model nested set seperti pada Gambar 8-3-2 dan tabelnya seperti di bawah ini.



Gambar 8-3-2.

```
CREATE TABLE org_nestedset (
  id INT PRIMARY KEY,
  kiri INT NOT NULL,
    INDEX(kiri),
  kanan INT NOT NULL, – CHECK (kanan > kiri)
    INDEX(kanan),
  nama VARCHAR(32) NOT NULL
);
mysql> SELECT * FROM org_nestedset;
+----+------+-------+-------------------------+
| id | kiri | kanan | nama                    |
+----+------+-------+-------------------------+
```

```
|  1 |    1 |    28 | Manajer Umum              |
|  2 |    2 |     9 | Manajer Sales & Marketing |
|  3 |   10 |    15 | Manajer Keuangan          |
|  4 |   16 |    21 | Manajer Teknis            |
|  5 |   22 |    25 | Manajer HRD               |
|  6 |   26 |    27 | Wakil Manajer Umum        |
|  7 |    3 |     6 | Staf Sales                |
|  8 |    7 |     8 | Staf Marketing            |
|  9 |    4 |     5 | Klerek Sales              |
| 10 |   11 |    14 | Staf Keuangan             |
| 11 |   12 |    13 | Klerek Keuangan           |
| 12 |   17 |    18 | Technical Support         |
| 13 |   19 |    20 | Teknisi                   |
| 14 |   23 |    24 | Staf HRD                  |
+----+------+-------+---------------------------+
14 rows in set (0.00 sec)
```

Untuk memberi nilai kiri dan kanan pada Gambar 8-3-2, saya mulai dari angka 1 untuk *kiri* milik root node ("Manajer Umum") lalu turun menelusuri sisi kiri tiap child node dan memberi angka pada *kiri* yang semakin membesar. Jika semua child node sudah habis, saya berpindah ke sisi kanan.

Dengan nested set, ini ada beberapa kemudahan yang diperoleh, terutama dalam berurusan dengan descendant nodes. Untuk mengambil semua descendant nodes milik *X*, kita hanya perlu melakukan uji: `kiri > kiriX AND kanan < kananX`. Dengan kata lain, mencari rentang *kiri-kanan* yang ada di dalam *kiriX-kananX*. Pengujian ini amat cepat dan sederhana karena hanya melibatkan operasi perbandingan angka.

Contoh, untuk mencari semua jabatan yang ada di bawah Manajer Sales & Marketing:

```
-- (asumsi kita sudah mengetahui nilai kiri dan kanan milik
Manajer
--    Sales & Marketing, yaitu 2 dan 9)
mysql> SELECT id, nama FROM org_nestedset WHERE kiri>2 AND
kanan<9;
```

Untuk mencari root node juga mudah, yaitu yang memiliki *kiri* paling kecil (atau *kanan* paling besar):

```
SELECT id, nama, MIN(kiri) FROM org_nestedset; -- atau MAX(kanan)
```

Yang sulit pada model nested set adalah memberikan nilai kiri dan kanan untuk pertama kali. Dan yang lebih sulit lagi adalah memaintain nilai kiri dan kanan agar sifatnya tetap sama. Setiap kali ingin insert node atau menghapus node, kita kadang harus menyesuaikan nilai *kiri* dan *kanan* seluruh baris pada tabel! Pada umumnya ini dilakukan dengan stored procedure.

Pembahasan lebih mendetil mengenai nested set dapat Anda baca pada buku *SQL for Smarties: Advanced SQL Programming* karangan Joe Celko.

Materialized Path

Cara ketiga menyimpan tree adalah dengan menyimpan path dari setiap node. Contoh:

```
CREATE TABLE org_mp (
  id INT PRIMARY KEY,
  path VARCHAR(255) NOT NULL, INDEX(path),
  nama VARCHAR(32) NOT NULL
); mysql> SELECT * FROM org_mp;
+----+-------------+--------------------------+
| id | path        | nama                     |
+----+-------------+--------------------------+
|  1 |             | Manajer Umum             |
|  2 | 0001        | Manajer Sales & Marketing |
|  3 | 0001        | Manajer Keuangan         |
|  4 | 0001        | Manajer Teknis           |
|  5 | 0001        | Manajer HRD              |
|  6 | 0001        | Wakil Manajer Umum       |
|  7 | 00010002    | Staf Sales               |
|  8 | 00010002    | Staf Marketing           |
|  9 | 000100020007 | Klerek Sales            |
| 10 | 00010003    | Staf Keuangan            |
| 11 | 000100030010 | Klerek Keuangan         |
| 12 | 00010004    | Technical Support        |
| 13 | 00010004    | Teknisi                  |
| 14 | 00010005    | Staf HRD                 |
+----+-------------+--------------------------+
14 rows in set (0.00 sec)
```

Pada contoh di atas kita membentuk path dari serangkaian id yang terdiri dari 4 digit. Ini berarti kita tidak mengharapkan jumlah node pada tree mencapai 10 ribu.

Dan kedalaman maksimum tree adalah sekitar 63 level (63 x 4 karakter = 252 karakter, ukuran maksimum VARCHAR adalah 255).

Menyimpan dengan cara materialized path memiliki beberapa keuntungan. Beberapa jenis query dapat diketahui dengan mudah. Misalnya, mencari root node cukup dengan mencari baris yang `path = ''`. Mengetahui kedalaman tree cukup dengan mencari *path* yang terpanjang, lalu membagi panjangnya dengan 4 dan menambah dengan 1 (12 / 4 + 1 = 3 + 1 = 4 level). Untuk mencari child node dari *X*, cukup dengan mencari dengan uji `WHERE path LIKE CONCAT(pathX,'____')`. Sementara untuk mencari semua descendant nodes dengan `WHERE path LIKE CONCAT(pathX, '%')`.

Cara materialized path juga memiliki keuntungan yaitu paling mudah dijaga integritasnya. Dengan trigger, kita dapat menjaga agar node yang ditambahkan haruslah parent node atau child dari node yang sudah ada (dengan mengecek path yang dimasukkan). Kita juga dapat menjaga bentuk tree misalnya hanya mengizinkan *N* buah child per node atau kedalaman maksimum *L* level. Saya tidak berkata bahwa metode AL atau NS tidak bisa dijaga integritasnya, karena dengan trigger *segalanya mungkin*, namun dengan MP pengujian integritas dapat dilakukan dengan hanya SELECT sederhana atau cukup melihat nilai atribut.

Metode yang Mana?

Metode adjacency list banyak dipakai oleh aplikasi-aplikasi PHP yang ada karena memang sederhana modelnya dan mirip dengan cara kita menyimpan struktur tree di bahasa pemrograman (menggunakan pointer). Di produk database seperti Oracle dan DB2 pun model tree ini amat umum dipakai karena adanya klausa khusus CONNECT BY yang khusus untuk tree sehingga dalam memilih descendant nodes kita tidak perlu melainkan UNION lagi.

Sayangnya, metode adjacency list bukanlah yang tercepat untuk tree yang berukuran amat besar, terutama dalam query-query yang mencari descendant nodes (bukan hanya child atau parent nodes) dikarenakan harus melakukan query rekursif/UNION.

Model nested set memiliki keuntungan utama dalam query yang melibatkan pencarian descendant nodes. Namun sulit dimaintain (diupdate dan didelete) karena nilai *kiri* dan *kanan* harus terus dijaga. Mengimplementasi nested set di MySQL pun amat sulit saat ini karena belum adanya dukungan terhadap stored procedure (di 4.x). Ditambah lagi, seorang pemula umumnya mengalami kesulitan agak lama sebelum memahami model ini.

Saya cenderung memilih model materialized path. Model ini meskipun cukup boros ruang disk namun memiliki banyak keuntungan. Mencari descendant nodes mudah dan tidak terlalu kalah cepat dengan model nested set, demikian juga mencari child/parent, mencari kedalaman, dan lain sebagainya. Kerugiannya hanyalah dalam hal pemakaian ruang disk dan harus membetulkan semua path descendant nodes manakala terjadi perpindahan node. Ruang disk semakin murah saat ini dan pada umumnya sebuah tree tidak terlalu sering berubah-ubah posisi nodenya. Namun perlu dipertimbangkan seandainya tree Anda bentuknya cukup tak lazim (misalnya: amat dalam hingga berpuluh-ratus level tapi amat "sempit" mendekati linear) karena pemborosan ruang disk yang terjadi akan cukup besar. MySQL pun saat ini belum mengizinkan kita mengindeks kolom BLOB dan TEXT dengan indeks B+Tree biasa sehingga untuk path hanya maksimum bisa disimpan di VARCHAR(255). Di database lain seperti Firebird dan PostgreSQL, kita bisa membuat tipe VARCHAR yang jauh lebih panjang.

8.4 Spatial

Fitur spatial didukung oleh MySQL sejak versi 4.1 dan terdapat juga pada beberapa produk database seperti PostgreSQL dan Oracle. Apakah fitur spatial itu? Spatial adalah dukungan untuk tipe-tipe data geometri/geografis seperti titik (koordinat), garis, persegipanjang, poligon, dan lain sebagainya. Sebetulnya untuk entiti-entiti seperti ini bisa saja dimodelkan dengan tipe data dasar database, misalnya titik dengan dua buah DOUBLE x, y; garis dengan dua buah titik x1, y1, x2, y2, dan seterusnya. Namun tipe data geografis ini memiliki keuntungan yaitu dapat diindeks menggunakan tipe indeks R-Tree. Dengan tipe data ini, kita dapat dengan cepat menjawab pertanyaan apakah sebuah titik berada dalam rentang lokasi tertentu, atau apakah sebuah persegipanjang bersentuhan dengan 1000 persegipanjang lain, dan lain sebagainya.

Membuat Kolom Bertipe Geometri

Untuk membuat tipe data geometri, kita menggunakan kata kunci GEOMETRY. Kolom bertipe ini dapat dipakai untuk menyimpan berbagai jenis bentuk geometri, mulai dari titik, garis, kurva, persegipanjang, dan lain sebagainya. Dan untuk mengindeksnya, kita gunakan kata kunci SPATIAL INDEX:

```
CREATE TABLE geom (g GEOMETRY NOT NULL, SPATIAL INDEX(g));
```

Mengisi Kolom

Untuk mengisi kolom dengan nilai geometri, kita perlu menggunakan fungsi GeomFromText() lalu mendefinisikan bentuk dan koordinat yang diinginkan. Ini

dikarenakan tipe data geometri disimpan dalam representasi khusus sementara kita memasukkan dalam representasi teks yang perlu dikonversi sebelum dimasukkan ke database.

```
INSERT INTO geom VALUES (GeomFromText('POINT(1 1)'));
INSERT INTO geom VALUES (GeomFromText('POINT(1 4)'));
INSERT INTO geom VALUES (GeomFromText('POINT(2 5)'));
INSERT INTO geom VALUES (GeomFromText('POINT(3 3)'));
INSERT INTO geom VALUES (GeomFromText('POINT(3 7)'));
INSERT INTO geom VALUES (GeomFromText('POINT(5 10)'));
```

Selain POINT(), Anda juga bisa memasukkan LINESTRING(), POLYGON(), dan lain sebagainya. Untuk lengkapnya silakan lihat manual MySQL.

Menampilkan Nilai Geometri

Untuk menampilkan nilai geometri, kita mengkonversikan dulu representasi internal kolom menjadi representasi teks dengan fungsi AsText().

```
mysql> SELECT AsText(g) FROM geom;
+-------------+
| AsText(g)   |
+-------------+
| POINT(1 1)  |
| POINT(1 4)  |
| POINT(2 5)  |
| POINT(3 3)  |
| POINT(3 7)  |
| POINT(5 10) |
+-------------+
6 rows in set (0.00 sec)
```

Mencari Nilai Geometri

Dengan spatial index, mencari nilai geometri yang memiliki hubungan tertentu dengan nilai geometri lain menjadi cepat. Contoh:

```
mysql> SELECT AsText(g) FROM geom WHERE
       MBRContains(GeomFromText('POLyGON((1 1,1 3,3 3,3 1,1
1))'), g);
+------------+
| AsText(g)  |
+------------+
```

```
| POINT(1 1) |
| POINT(3 3) |
+------------+
2 rows in set (0.05 sec)
```

Query di atas mencari titik-titik mana saja yang ada di dalam rentang persegipanjang dengan koordinat (1,1) s.d. (3,3). Tentu saja bukan hanya titik yang dapat dites di sini melainkan juga bentuk-bentuk geometri lain jika kebetulan berada di dalam rentang persegipanjang tersebut.

Dalam kehidupan nyata, fitur spatial bisa dipakai misalnya untuk mencatat koordinat kota atau alamat lalu kita dengan cepat bisa mencari semua pelanggan yang jaraknya tak lebih dari 100km dari kota Bandung, atau yang tinggal berdekatan satu sama lain. Aplikasi lainnya tentunya dalam bidang pemetaan, yang memang sehari-hari berkutat dengan koordinat. Kita bisa mencatat rute perjalanan, rute kereta api, detail jalan dengan kurva lalu nanti mengecek berapa panjang sebuah rute, apakah rute A bersentuhan dengan rute B, dan lain sebagainya.

8.5 Full-Text Indexing

Jika nilai geometri dapat diindeks dengan indeks khusus (SPATIAL INDEX) maka kolom TEXT juga dapat diindeks dengan indeks khusus bernama FULLTEXT INDEX. Jenis FULLTEXT ini akan mengindeks semua kata yang ada dalam teks agar kita dapat melakukan pencarian sebuah kata dengan jauh lebih cepat dibandingkan cara `WHERE kolom LIKE '%kata_yang_dicari%'`.

Membuat Indeks FULLTEXT

Untuk membuat indeks pada sebuah kolom full text:

```
CREATE TABLE ft1 (
  ...
  nama_kolom TEXT,
  FULLTEXT INDEX (nama_kolom)
);
Untuk membuat indeks pada gabungan beberapa kolom TEXT sekaligus:
CREATE TABLE ft2 (
  ...
  teks1 TEXT,
  teks2 TEXT,
  FULLTEXT INDEX (teks1, teks2)
);
```

Pencarian FULLTEXT

Untuk melakukan pencarian, gunakan operator MATCH:

```
SELECT ... FROM ft1 WHERE
  MATCH (nama_kolom) AGAINST ('kata kunci');
SELECT ... FROM ft2 WHERE
  MATCH (teks1, teks2) AGAINST ('kata kunci');
```

Beberapa catatan: Pertama, secara default MySQL memiliki perbendaharaan kata perkecualian ("stopword") yaitu kata-kata dalam bahasa Inggris yang terlalu umum dan tidak akan diindeks, seperti `the`, `is`, `am`, dan lain sebagainya. Daftar lengkapnya ada di manual MySQL. Kedua, MySQL juga kemungkinan tidak akan menampilkan hasil jika jumlah baris di tabel masih terlalu sedikit (mis: baru 2 hingga 5 record), dikarenakan adanya "ambang 50%". Jika sebuah kata terdapat di lebih dari separuh dokumen yang ada, maka kata tersebut akan dikeluarkan dari kata kunci pencarian. Ketiga, MySQL memiliki opsi pencarian boolean dengan sintaks: `MATCH (...) AGAINST('+kata1 -kata2' IN BOOLEAN MODE)`. Dalam mode ini, MySQL akan mencari semua dokumen yang mengandung *kata1* dan *tidak* mengandung *kata2*.

Full-text indexing amat berguna jika Anda memiliki banyak teks di database dan ingin melakukan pencarian yang lebih cepat dari sekedar pencarian menggunakan LIKE atau REGEXP (yang bisa *amat sangat* lambat). Full-text index sering digunakan dalam aplikasi seperti forum atau CMS (contohnya bisa dilihat di Bab 9).